

## Capítulo DOIS

### Herança e Polimorfismo

---

#### Objetivos do Exame

- Implementar herança, incluindo modificadores de visibilidade e composição.
  - Sobrescrever os métodos hashCode, equals e toString da classe Object.
  - Implementar polimorfismo.
  - Desenvolver código que utiliza classes e métodos abstratos.
- 

#### Herança

No núcleo de uma linguagem orientada a objetos está o conceito de herança.

Em termos simples, herança refere-se a uma relação **É-UM** (*IS-A*) onde uma classe (chamada superclasse) fornece atributos e métodos comuns para classes derivadas ou mais especializadas (chamadas subclasses).

Em Java, uma classe só pode herdar de uma única superclasse (**herança singular**). Claro, a única exceção é java.lang.Object, que **não tem superclasse**. Essa classe é a superclasse de todas as classes.

A palavra-chave extends é usada para especificar essa relação. Por exemplo, um martelo **É-UMA** ferramenta, então podemos modelar isso como:

```
java                                                                    Copiar  Editar

class Tool {
    public int size;
}

class Hammer extends Tool {
}
```

Como size é um atributo public, ele é herdado por Hammer:

```
java                                                                    Copiar  Editar

Hammer hammer = new Hammer();
hammer.size = 10;
```

Do capítulo anterior, sabemos que apenas membros private e com visibilidade padrão (default), quando a subclasse está definida em um pacote diferente da superclasse, **não são herdados**.

---

Um atributo ou método é herdado com o mesmo nível de visibilidade definido na superclasse. No entanto, no caso de métodos, **você pode torná-los mais visíveis**, mas **não pode torná-los menos visíveis**:

```
java Copiar Editar

class Tool {
    public int size;
    public int getSize() { return size; }
}

class Hammer extends Tool {
    private int size; // Sem problema!
    // Erro de compilação
    private int getSize() { return size; }
}
```

Não há problema para atributos porque estamos criando um **NOVO** atributo em Hammer que **oculta** aquele herdado de Tool, quando o nome é o mesmo.

---

### Aqui estão as coisas que você pode fazer em uma subclasse:

- Atributos herdados podem ser usados diretamente, como qualquer outro.
  - Um atributo pode ser declarado na subclasse com o mesmo nome de um na superclasse, ocultando-o.
  - Novos atributos que não existem na superclasse podem ser declarados na subclasse.
  - Métodos herdados podem ser usados diretamente como estão.
  - Um novo método de instância pode ser declarado na subclasse com a mesma assinatura de um na superclasse, sobrescrevendo-o.
  - Um novo método static pode ser declarado na subclasse com a mesma assinatura de um na superclasse, ocultando-o.
  - Novos métodos que não existem na superclasse podem ser declarados na subclasse.
  - Um construtor pode ser declarado na subclasse que invoca o construtor da superclasse, de forma implícita ou usando a palavra-chave `super`.
- 

Portanto, para métodos, **reduzir sua visibilidade não é permitido** porque eles são tratados de maneira diferente. Em outras palavras, os métodos são **sobrescritos** ou **sobrecarregados**.

Além disso, pense sobre isso. Por causa do encapsulamento, os atributos devem estar escondidos, mas com métodos, se uma subclasse **não tiver um método** da superclasse, **a subclasse não pode ser usada onde a superclasse é usada**.

Isso é chamado de **princípio da substituição de Liskov**, o qual é importante no polimorfismo, e revisaremos depois de falar sobre **métodos sobrescritos e sobrecarregados**.

---

Implementar uma interface é, de certa forma, um tipo de herança porque compartilham algumas características comuns. Mas ao fazer isso, a relação se torna **TEM-UM** (*HAS-A*).

Falaremos mais sobre isso no Capítulo 4.

---

### Sobrecarga e Sobrescrita

A diferença entre **sobrecarga** (*overloading*) e **sobrescrita** (*overriding*) está muito relacionada com **assinaturas de métodos**.

Em poucas palavras, a **assinatura de um método** é o nome do método e a lista de seus parâmetros (**tipos e número de parâmetros incluídos**).

Observe que **tipos de retorno não estão incluídos** nessa definição.

Falamos de **sobrecarga** quando um método muda a assinatura do método, alterando a lista de parâmetros de outro método (que pode ser herdado), mantendo o mesmo nome.

```
java Copiar Editar

class Hotel {
    public void reserveRoom(int rooms) { ... }
}

class ThreeStarHotel extends Hotel {
    // Sobrecarga do método #1
    public void reserveRoom(List<Room> rooms) {
        ...
    }
    // Sobrecarga do método #2
    public void reserveRoom(
        int rooms, int numberPeople) {
        ...
    }
}
```

Mudar **apenas o tipo de retorno** gerará um erro de compilação:

```
java Copiar Editar

class ThreeStarHotel extends Hotel {
    // Erro de compilação, reserveRoom é visto como duplicado
    public void reserveRoom(List<Room> rooms) {
        ...
    }
    public boolean reserveRoom(List<Room> rooms) {
        ...
    }
}
```

Exceções na cláusula throws **não são consideradas** na sobrecarga, então novamente, mudar apenas a lista de exceções gerará erro de compilação:

```
java Copiar Editar

class ThreeStarHotel extends Hotel {
    // Erro de compilação, reserveRoom é visto como duplicado
    public void reserveRoom(List<Room> rooms)
        throws RuntimeException {
        ...
    }
    public boolean reserveRoom(List<Room> rooms)
        throws NullPointerException {
        ...
    }
}
```

Quando um método sobrecarregado é chamado, o compilador tem que decidir **qual versão do método será chamada**.

O primeiro candidato óbvio é chamar o método que **exatamente corresponde ao número e tipo dos argumentos**.

Mas o que acontece quando **não há correspondência exata**?

A regra a lembrar é que o Java buscará a **correspondência mais próxima primeiro** (isto significa: um tipo maior, uma superclasse, um tipo com autoboxing ou um tipo mais específico).

```
java
class Print {
    static void printType(short param) {
        System.out.println("short");
    }
    static void printType(long param) {
        System.out.println("long");
    }
    static void printType(Integer param) {
        System.out.println("Integer");
    }
    static void printType(CharSequence param) {
        System.out.println("CharSequence");
    }

    public static void main(String[] args) {
        byte b = 1;
        int i = 1;
        Integer integer = 1;
        String s = "1";

        printType(b);
        printType(i);
        printType(integer);
        printType(s);
    }
}
```

Saída:

```
vbnet
short
long
Integer
CharSequence
```

Na primeira chamada de método, o tipo do argumento é byte. Não há método que receba byte, então o tipo maior mais próximo é short.

Na segunda chamada, o tipo do argumento é int. Não há método que receba int, então o tipo maior mais próximo é long (observe que isso tem maior precedência que Integer).

Na terceira chamada, o tipo do argumento é Integer. Existe um método que recebe Integer, então este é chamado.

Na última chamada, o tipo do argumento é String. Não há método que receba String, então a superclasse mais próxima é CharSequence.

---

Se **não puder encontrar uma correspondência** ou **se o compilador não conseguir decidir** porque a chamada é ambígua, um erro de compilação será lançado. Por exemplo, considerando a classe anterior, o seguinte causará erro porque **não existe tipo maior que double e não pode ser autoboxado para Integer**:

```

java                                                                    Copiar  Editar

// Não encontra uma correspondência
double d = 1.0;
printType(d);

```

O exemplo a seguir é uma chamada ambígua, assumindo os métodos:

```

java                                                                    Copiar  Editar

static void printType(float param, double param2) {
    System.out.println("float-double");
}
static void printType(double param, float param2) {
    System.out.println("double-float");
}

```

```

java                                                                    Copiar  Editar

// Chamada ambígua
printType(1, 1);

```

**Construtores de uma classe também podem ser sobrecarregados.**

Na verdade, você pode chamar um construtor a partir de outro usando a palavra-chave `this`:

```

java                                                                    Copiar  Editar

class Print {
    Print() {
        this("Chamando com argumento padrão");
    }
    Print(String s) {
        System.out.println(s);
    }
}

```

Falamos de sobrescrita (*overriding*) quando a **assinatura do método é a mesma**, mas por algum motivo, queremos redefinir um **método de INSTÂNCIA** na subclasse.

```

java                                                                    Copiar  Editar

class Hotel {
    public void reserveRoom(int rooms) {
        ...
    }
}
class ThreeStarHotel extends Hotel {
    // Sobrescrita de método
    public void reserveRoom(int rooms) {
        ...
    }
}

```

Se um método static com a mesma assinatura de um método static na superclasse for definido na subclasse, então o método é **ocultado** (*hidden*), em vez de sobrescrito.

---

Existem algumas regras ao sobrescrever um método:

- O modificador de acesso deve ser o mesmo ou com **mais visibilidade**:

```
java Copiar Editar

class Hotel {
    public void reserveRoom(int rooms) {
        ...
    }
}

class ThreeStarHotel extends Hotel {
    // Erro de compilação
    protected void reserveRoom(int rooms) {
        ...
    }
}
```

O tipo de retorno deve ser o mesmo ou um subtipo:

```
java Copiar Editar

class Hotel {
    public Integer reserveRoom(int rooms) {
        ...
    }
}

class ThreeStarHotel extends Hotel {
    // Erro de compilação
    public Number reserveRoom(int rooms) {
        ...
    }
}
```

As exceções na cláusula throws devem ser as mesmas, menores ou **subclasses** dessas exceções:

```
java Copiar Editar

class Hotel {
    public void reserveRoom(int rooms)
        throws IOException {
        ...
    }
}

class ThreeStarHotel extends Hotel {
    // Erro de compilação
    public void reserveRoom(int rooms) throws Exception {
        ...
    }
}
```

A sobrescrita é um conceito crítico no polimorfismo, mas antes de tratar desse tópico, vejamos alguns métodos importantes da classe `java.lang.Object`, que, na maioria das vezes, precisaremos sobrescrever.

---

## Métodos da Classe Object

Em Java, todos os objetos herdam de `java.lang.Object`.

Essa classe possui os seguintes métodos que podem ser sobrescritos (redefinidos):

```
java                                                                    Copiar  Editar
protected Object clone() throws CloneNotSupportedException
protected void finalize() throws Throwable
public int hashCode()
public boolean equals(Object obj)
public String toString()
```

Os métodos mais importantes — aqueles que **quase sempre você vai querer redefinir** — são:

- hashCode
- equals
- toString

### **public int hashCode()**

Retorna um valor de código hash para o objeto. O valor retornado deve obedecer ao seguinte contrato:

- Sempre que for invocado no mesmo objeto mais de uma vez durante a execução de uma aplicação Java, o método hashCode **deve retornar consistentemente o mesmo valor inteiro**, desde que nenhuma informação usada nas comparações com equals seja modificada.  
Esse valor **não precisa permanecer consistente entre execuções** diferentes da mesma aplicação.
- Se dois objetos são iguais de acordo com o método equals(Object), então chamar hashCode em cada um **deve retornar o mesmo valor inteiro**.
- Não é obrigatório que, se dois objetos **não forem iguais** segundo equals(Object), seus métodos hashCode retornem inteiros diferentes.  
Entretanto, **produzir resultados distintos para objetos não iguais pode melhorar a performance de tabelas hash (hash tables)**.

---

### **public boolean equals(Object obj)**

Indica se outro objeto é **igual ao objeto que chama o método**.

É necessário sobrescrever o método hashCode sempre que equals for sobrescrito, uma vez que o contrato de hashCode exige que objetos iguais tenham o mesmo código hash.

Este método é:

- **Reflexivo**: para qualquer valor de referência não nulo x, x.equals(x) deve retornar true.
- **Simétrico**: para quaisquer valores de referência não nulos x e y, x.equals(y) deve retornar true se e somente se y.equals(x) retornar true.
- **Transitivo**: para quaisquer valores de referência não nulos x, y e z, se x.equals(y) for true e y.equals(z) for true, então x.equals(z) deve retornar true.
- **Consistente**: para quaisquer valores de referência não nulos x e y, múltiplas invocações de x.equals(y) devem retornar consistentemente true ou false, desde que nenhuma informação usada na comparação seja modificada.
- Para qualquer valor de referência não nulo x, x.equals(null) deve retornar false.

---

### **public String toString()**

Retorna uma representação em String do objeto.

O método `toString` da classe `Object` retorna uma string que consiste no nome da classe da qual o objeto é instância, seguido pelo caractere `@` e pela representação hexadecimal não assinada do código hash do objeto.

---

Para sobrescrever esses métodos, basta seguir as regras gerais de sobrescrita:

- O modificador de acesso deve ser o mesmo ou mais acessível
- O tipo de retorno deve ser o mesmo ou uma subclasse
- O nome do método deve ser o mesmo
- A lista de tipos de argumentos deve ser a mesma
- As mesmas exceções (ou suas subclasses) podem ser lançadas

**Em poucas palavras:** defina o método exatamente como ele aparece na classe `java.lang.Object`.

## Polimorfismo

Polimorfismo é a capacidade de um objeto variar seu comportamento com base em seu tipo.

Isso é melhor demonstrado com um exemplo:

```
java Copiar Editar

class HumanBeing {
    public void dress() {
        System.out.println("Dressing a human being");
    }
}
class Man extends HumanBeing {
    public void dress() {
        System.out.println("Put on a shirt");
        System.out.println("Put on some jeans");
    }
}
class Woman extends HumanBeing {
    public void dress() {
        System.out.println("Put on a dress");
    }
}
class Baby extends HumanBeing {
    public void dress() {
        System.out.println("I don't know how to dress!");
    }
}
```

E agora, vamos criar alguns seres humanos para ver o polimorfismo em ação:

```
java Copiar Editar

HumanBeing[] someHumans = new HumanBeing[3];
someHumans[0] = new Man();
someHumans[1] = new Woman();
someHumans[2] = new Baby();

for(int i = 0; i < someHumans.length; i++) {
    someHumans[i].dress();
    System.out.println();
}
```



## Saída:

```
vbnet
Put on a shirt
Put on some jeans

Put on a dress

I don't know how to dress!
```

Mesmo usando `HumanBeing`, a JVM decide **em tempo de execução** qual método chamar com base no **tipo do objeto atribuído**, e não no tipo da referência da variável.

Isso é chamado de **invocação de método virtual**, um nome chique para sobrescrita.

A sobrescrita também é conhecida como **polimorfismo dinâmico** porque o tipo do objeto é decidido em **tempo de EXECUÇÃO**.

Por contraste, a sobrecarga é chamada de **polimorfismo estático** porque é resolvida em **tempo de COMPILAÇÃO**.

---

## Classes e Métodos Abstratos

Se examinarmos o exemplo anterior, acredito que concordaremos que a implementação do método `dress()` na classe `HumanBeing` **não soa exatamente correta**.

Na maioria das vezes, trabalharemos com algo mais concreto, como um `Man` ou uma `Woman`, portanto, **não há necessidade de instanciar diretamente a classe `HumanBeing`**.

No entanto, uma abstração comum dessas classes pode ser útil.

Usar uma **classe abstrata** (ou método abstrato) é a melhor opção para modelar esses casos.

---

## Regras de classes abstratas:

Classes abstratas **NÃO PODEM ser instanciadas**, apenas estendidas.

São declaradas com a palavra-chave `abstract`:

```
java
abstract class AClass { }
```

Métodos abstratos são declarados **SEM implementação** (sem corpo), como isto:

```
java
abstract void AMethod();
```

Então, no exemplo anterior, é melhor modelar toda a classe `HumanBeing` como `abstract` para que ninguém possa usá-la diretamente:

```
java
abstract class HumanBeing {
    public abstract void dress();
}
```

Agora, o seguinte causará **erro de compilação**:

```
java Copiar Editar

HumanBeing human = new HumanBeing();
```

E isso faz sentido: **não há garantias de que uma classe abstrata terá todos os seus métodos implementados**. Chamar um método não implementado seria um fracasso épico.

---

### Regras ao trabalhar com métodos e classes abstratas:

- A palavra-chave `abstract` **só pode ser aplicada a classes ou métodos não estáticos**.

```
java Copiar Editar

abstract class AClass {
    // Erro de compilação
    public static abstract void AMethod();
}
```

Uma classe abstrata **não precisa declarar métodos abstratos** para ser abstrata.

```
java Copiar Editar

abstract class AClass { } // Sem problema
```

Se uma classe incluir métodos abstratos, **ela mesma deve ser declarada como abstrata**.

```
java Copiar Editar

class AClass { // Erro de compilação
    public abstract void AMethod();
}
```

Se a subclasse de uma classe abstrata **não fornecer implementação para todos os métodos abstratos, ela também deve ser declarada como abstrata**.

```
java Copiar Editar

// Erro de compilação
class Man extends HumanBeing { }
```

Métodos de uma interface são considerados abstratos, portanto, uma classe abstrata que implementa uma interface **pode implementar alguns ou nenhum dos métodos da interface**.

```
java Copiar Editar

// Sem problema
abstract class AClass implements Runnable {}
```

---

### Pontos-Chave

- **Herança** refere-se a uma relação **É-UM**, onde uma classe (superclasse) fornece atributos e métodos comuns para classes derivadas ou mais especializadas (subclasses).

### O que você pode fazer em uma subclasse:

- Usar atributos herdados diretamente.
- Declarar um atributo com o mesmo nome de um da superclasse, ocultando-o.

- Declarar novos atributos que não existem na superclasse.
  - Usar métodos herdados diretamente como estão.
  - Declarar um novo método de instância com a mesma assinatura, sobrescrevendo-o.
  - Declarar um novo método estático com a mesma assinatura, ocultando-o.
  - Declarar novos métodos não existentes na superclasse.
  - Declarar um construtor que invoque o da superclasse, de forma implícita ou usando `super`.
- 

- A **assinatura de um método** é o nome do método + lista de parâmetros (tipo e quantidade).  
Tipos de retorno **não estão incluídos**.
  - **Sobrecarga** ocorre quando um método muda a lista de parâmetros de outro, mantendo o mesmo nome.
  - **Sobrescrita** ocorre quando a assinatura é a mesma, mas queremos redefinir o método na subclasse.
  - Os métodos mais importantes de `java.lang.Object` que geralmente devem ser redefinidos são:
    - `public int hashCode()`
    - `public boolean equals(Object obj)`
    - `public String toString()`
- 

- Com o **polimorfismo**, as subclasses podem definir seus próprios comportamentos (diferentes dos da superclasse), e a JVM chamará o método apropriado do objeto.  
Esse comportamento é chamado de **invocação de método virtual**.
- **Classes abstratas NÃO podem ser instanciadas**, apenas estendidas.  
Métodos abstratos são declarados **sem corpo**.
- `abstract` só pode ser aplicado a classes ou métodos não estáticos.
- Uma classe abstrata **não precisa** declarar métodos abstratos para ser abstrata.
- Se uma classe possui métodos abstratos, ela **deve ser declarada como abstrata**.
- Se uma subclasse não implementar todos os métodos abstratos, ela também deve ser abstrata.
- Métodos de uma interface são abstratos; logo, uma classe abstrata que implementa uma interface pode implementar alguns ou nenhum desses métodos.

## Autoavaliação

### 1. Dado:

```
java Copiar Editar

public class Question_2_1 {
    protected int id;
    protected String name;

    protected boolean equals(Question_2_1 q) {
        return this.name.equals(q.name);
    }

    public static void main(String[] args) {
        Question_2_1 q1 = new Question_2_1();
        Question_2_1 q2 = new Question_2_1();
        q1.name = "q1";
        q2.name = "q1";

        if(q1.equals((Object)q2)) {
            System.out.println("true");
        } else {
            System.out.println("false");
        }
    }
}
```

Qual é o resultado?

- A. true
- B. false
- C. A compilação falha
- D. Uma exceção ocorre em tempo de execução

2. Qual dos seguintes é um método da classe `java.lang.Object` que pode ser sobrescrito?

- A. `public String toString(Object obj)`
- B. `public int equals(Object obj)`
- C. `public int hashCode(Object obj)`
- D. `public int hashCode()`

### 3. Dado:

```
java Copiar Editar

public class Question_2_3 {
    public static void print(Integer i) {
        System.out.println("Integer");
    }
    public static void print(Object o) {
        System.out.println("Object");
    }
    public static void main(String[] args) {
        print(null);
    }
}
```

Qual é o resultado?

- A. Integer
- B. Object
- C. A compilação falha
- D. Uma exceção ocorre em tempo de execução

#### 4. Dado:

```
java Copiar Editar

class SuperClass {
    public static void print() {
        System.out.println("Superclass");
    }
}

public class Question_2_4 extends SuperClass {
    public static void print() {
        System.out.println("Subclass");
    }

    public static void main(String[] args) {
        print();
    }
}
```

Qual é o resultado?

- A. Superclass
- B. Subclass
- C. A compilação falha
- D. Uma exceção ocorre em tempo de execução

#### 5. Dado:

```
java Copiar Editar

abstract class SuperClass2 {
    public static void print() {
        System.out.println("Superclass");
    }
}

class SubClass extends SuperClass2 {}

public class Question_2_5 extends SuperClass {
    public static void main(String[] args) {
        SubClass subclass = new SubClass();
        subclass.print();
    }
}
```

Qual é o resultado?

- A. Superclass
- B. A compilação falha porque uma classe abstrata não pode ter métodos static
- C. A compilação falha porque SubClass não implementa o método print()
- D. A compilação falha porque SubClass não possui um método print()
- E. Uma exceção ocorre em tempo de execução

## 6. Dado:

```
java Copiar Editar

abstract class SuperClass3 {
    public void print() {
        System.out.println("Superclass");
    }
}

public class Question_2_6 extends SuperClass3 {
    public void print() {
        System.out.println("Subclass");
    }
    public static void main(String[] args) {
        Question_2_6 q = new Question_2_6();
        ((SuperClass3)q).print();
    }
}
```

Qual é o resultado?

- A. Superclass
- B. Subclass
- C. A compilação falha
- D. Uma exceção ocorre em tempo de execução